ELECTRONIC PUBLISHING, VOL. 6(3), 159–170 (SEPTEMBER 1993)

Object-orientation and extensibility in a font-scaler

BEAT STAMM

Institute for Computer Systems
Swiss Federal Institute of Technology
CH–8092 Zürich, Switzerland

e-mail: stamm@inf.ethz.ch

SUMMARY

Today's font-scalers generate screenfonts with acceptable quality on-the-fly from a generic font representation. However, as closed systems they discourage the integration of separate solutions to different aspects of font-scaling. This paper illustrates an *object-oriented approach* **that allows for both** *contour and rendering independence***. Refined solutions can be packaged separately into intelligent contour and rendering objects. The approach results in a** *small and efficient font-scaling system* **that masters complexity by concept rather than industriousness.**

KEY WORDS Font representation Medium and low-resolution font-scaling Object-orientation Contour and rendering independence

1 INTRODUCTION

Font-scaling denotes the process of producing bitmapped fonts from a generic font representation. Each bitmapped font comprises a particular choice of pixels that mimic as closely as possible an individual type size at a given device resolution. This choice is the result of inevitable rounding operations that are performed eventually in the process of font-scaling. With these rounding operations the overall scaling function becomes non-linear.

Ignoring this non-linearity, naive font-scalers are prone to introduce irregularities not present in the original font, such as unequal stem widths or asymmetric serifs. Likewise, subtle near-regularities, such as reference line overlaps or optical corrections to the thicknesses of strokes, are at risk of being enlarged disproportionately. To preserve the regularity properties of a font, we have proposed a hierarchy for decomposing characters into their constituent parts (*glyphs*, *contours*, *knots*, and *numbers*) [1]. Using this particular decomposition, we illustrate in [2] how the regularity of coarsely scaled fonts is attained dynamically.

One may object that implementing a scaler for hierarchically structured fonts is easier said than done, all the more considering the multitude of artistic aspects of font-scaling. This need not be so necessarily, if the implementation is done in object-oriented programming (OOP) style. Starting out from the hierarchy, Section 2 of this paper introduces the basics of an extensible engineering framework for font-scaling. Section 3 proceeds with illustrating several advanced extensions of the base system and leads to suggestions for future refinements. Section 4 concludes with a summary, a brief comparison with existing approaches, and some performance figures about the prototype.

CCC 0894–3982/93/030159–12

Received 1 September 1993
Revised 3 December 1993

2 THE BASIC FRAMEWORK

2.1 A hierarchical structure

In the proposed hierarchy a component is either an *immediate component*, or it is composed of other components, which themselves may be composed of other components, etc. Examples:

$$\begin{aligned} \texttt{rightStemEdge} &= \texttt{leftStemEdge} + \texttt{stemWidth} \\ \texttt{strokeThickness} &= \texttt{regularThickness} \pm \texttt{opticalCorrection} \\ \texttt{apex} &= \texttt{meanline} + \texttt{overshoot} - \texttt{strokeThickness} \end{aligned}$$

These examples reflect in detail the nature of three *number* parts. Their constituent parts themselves are scaled and rounded separately before they are added to (or subtracted from) one another. This re-establishes the regularity of scaled numbers by concept.

On the next higher level of the hierarchy, *knots* aggregate pairs of numbers that assume the role of coordinates. Analogously, the knots may be the result of the separate addition (or subtraction) of other knots that assume the role of vectors. On the second-to-highest level, *contours* contain lists of elements referring to knots that are mapped onto the geometry of the underlying graphical object. Contours are digitised separately before they are concatenated. Open contours are concatenated to more complex contours, the sign denoting their orientation, and eventually result in a glyph (closed contour). On the top level, *characters* are created out of one or more glyphs or counterforms.

In this way, each level of the hierarchy is represented by a particular class of constituent parts (glyphs, contours, knots, and numbers). Each part is composed of one or more parts from the next lower level of the hierarchy. Alternatively, each of these parts may be a combination of other parts of the same hierarchical level. This distinction between terminal and non-terminal entities allows for any depth of recursion in the characters' blueprints. Nevertheless, the font-scaler need not be permanently aware of this recursiveness, as will be shown in section 2.3.

2.2 Attributed components

Immediate components may occur in different varieties. A list of knots gives the essential ingredients of a contour, but not the particular variety of contour (polygon, Bézier curve, etc.) on to whose geometry the knots are mapped. Likewise, some numbers must be unconditionally safeguarded against vanishing under coarse scaling, while others may do so quite safely. Therefore, the suggested hierarchy proposes the use of *attributes* to specify the particular variety of the component. Examples:

NUMBER `stemWidth = 33↑`
{ *a number component which must not vanish under coarse scaling* }
NUMBER `opticalCorrection = 3`
{ *a number component which may (and will) vanish under coarse scaling* }
GLYPH `stem` = CLOSED POLYGON [... `capHeight, stemWidth ...`]
{ *a predefined contour class, provided with the font-scaler* }
GLYPH `criticalQuadrant = GraphicExt.StopGapCurve` [...]
{ *an extended contour class, added to the font-scaler later on* }

What is important is to provide the possibility of introducing further attributes on demand, such as the `GraphicExt.StopGapCurve` in the above example. This extends the palette of primitives and may give a means of talking not only about outlines but also about appearance (cf. Section 3). In the end, this allows for any number of attributes in the characters' blueprints. The next section shows why, nevertheless, the font-scaler need not know all the varieties simultaneously.

2.3 Polymorphism in a font representation

The preceding section calls for homogeneous treatment of all kinds of varieties of one and the same basic entity. This is a typical case of the application of *polymorphism*. The functionality of the entities is the same, but their implementation is not: both polygons and Bézier curves are contours suitable for defining the boundaries of glyphs and characters (as are natural splines and circular arcs), but their scan-conversion algorithms differ from contour class to contour class. Analogously, the response of numbers to scaling varies along with the attribute given to the individual number, but the common functionality (to be scaleable numbers at all) does not.

The homogeneous view on to varieties of the same component can be expanded smoothly to a variety that refers to a sum of other components of the same basic class. By dint of polymorphism, these 'other components' by themselves can be either one of the varieties of the immediate component, or again one that refers to other components. With this, both the alternative parts and the assembly thereof are unified — irrespective of the numerous alternatives and the recursive depth of the assembly!

The following example illustrates these facts in practice. The program fragments are given in a notation similar to Pascal or Modula-2, with the addition of a class construct and methods. The intention in doing so is to express object-oriented programming (OOP) style (polymorphism or inheritance and late binding) without referring to particular hybrid languages.

```
Contour = CLASS
    METHOD ScanConvert( ... ) { an abstract or virtual method };
END;

Polygon = CLASS [super = Contour] { extends the Contour class above }
    verts: LIST OF Vertex;
    METHOD ScanConvert( ... ) { overwrite with concrete scan-conversion algorithm };
END;

BezierCurve = CLASS [super = Contour] { as for Polygon above }
    polys: LIST OF BezierPoly;
    METHOD ScanConvert( ... );
END;

Concatenation = CLASS [super = Contour]
    components: LIST OF Contour;
    METHOD ScanConvert( ... ) { overwrite with an iterator over components };
END;
```

This example forms a class hierarchy of contours. The other levels of the component hierarchy, the glyphs, the knots, and especially the numbers, form analogous class hierarchies.

By now, the proposed structuring of a font into components has become equivalent to building up in main memory a *polymorphic directed acyclic graph* (PolyDAG). Eventually, this PolyDAG has to be externalised (stored to a disk file) and internalised (loaded from a disk file) again. Benefits of OOP style are conveniently used to address this topic as well. Szyperski [3] suggests a generic method for loading and storing PolyDAGs with a mapping mechanism that preserves referential transparency. But this topic is not considered specific to font-related problems, and hence is not subjected to further analysis here.

The benefits of OOP style with respect to a font representation are twofold. First, recursiveness is straightened out in a way that is transparent to the caller. The caller does not have to be permanently aware of it. Second, the multitude of varieties is completely homogenised, again transparently. This encourages the *ad hoc* integration of independent solutions to different aspects of font-scaling. Object-orientation and extensibility have paved the way to a concise, yet comprehensive font-scaler.

2.4 Digitising vs. rendering

Font-scaling is likely not to be well understood without a sound background in the methodology of two-dimensional graphics. Without knowing all kinds of inconspicuous idiosyncrasies of digitising algorithms – even of a Bresenham-type routine for straight lines – a font-scaler may be prone to cure symptoms, rather than eradicating their causes. A first series of experiences we have made with two-dimensional graphics are captured in a graphics library for drawing lines and curves [4].

Based upon these premises, we developed an editor for outline fonts which was later to become the successor of the software described in [5]. Careful analysis of the conclusions drawn from using the first prototype of this editor has evolved the graphics library from a collection of procedures into a fully-grown object-oriented toolbox [6]. Its generality, despite its tiny size, takes a few steps beyond what is required for a font scaler or editor. It may serve very well as a basis for a general-purpose illustrator.

The outstanding result of this toolbox is a strict decoupling of *intelligent rendering tools* both from the graphical objects to be digitised, and from the raster devices on which the objects are to be rendered. In this toolbox, the polygon or circle objects do not outline or fill themselves directly. Rather, they serialise digitised coordinate pairs (the vertices of a rectilinear region), into an abstract drawing tool or renderer. Concrete renderers have the intelligence to determine which pixels are involved in rendering the digital input as outlined or solid object (Figure 1).

The pixels in turn are forwarded to an abstract raster device class, concrete derived classes of which assume the role of device drivers.

This strict decoupling is supported also in the PolyDAG that models a font. To do so, the scan-convert methods are parametrized with the rendering tool to be used.

```
Contour = CLASS
    METHOD ScanConvert (using: RenderingTool);
END;
```

With respect to a font-scaler, this offers three advantages. First, it eases the integration of other contour classes. This helps in meeting the demands of a foreign font format, or the preferences of a type designer, and it encourages non-standard scan-converters for the intrinsic contours. An example of a non-standard scan-converter is given in subsection 3.1.

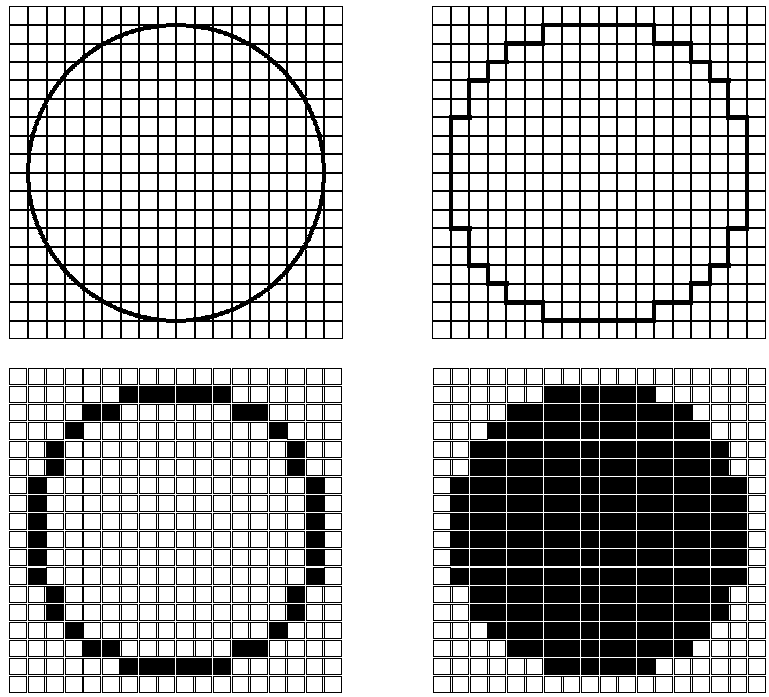

Figure 1. A circle (top left) and its rectilinear region (top right), rendered in outlined (bottom left) and filled style (bottom right)

Second, it opens the door to the smooth supply of alternative renderers. This avoids superfluous distinctions between outlined and filled characters, and it invites the devising of unconventional renderers. An example of rendering the same characters with different tools is given in subsection 3.3. Third, all contour classes added can be rendered readily with all current and future renderers and, vice-versa, all renderers contributed can be applied instantly to all current and future contour classes without touching the respective scan-converters. With this, provisions have been made for both *contour* and *rendering independence*.

3 ADVANCED USE OF EXTENSIBILITY

3.1 Non-standard scan-conversion

Subsection 2.2 mentions an attribute whose application to a number preserves the number from vanishing under coarse scaling. It is easy to see that this simple mechanism prevents dropouts in orthogonal rectangles (upright stems, crossbars) and in parallelograms (diagonal stems, with the left and right edges refering to the same edge component). For simple curvilinear glyphs this works as well, if the shapes are somewhat geometrical ones with almost uniform thicknesses of stroke, such as in the font Helvetica.

As soon as a font originates in handwriting with a broad-nibbed pen, even the font Times, the local extrema of the inner and the outer outlines may be dislocated from one

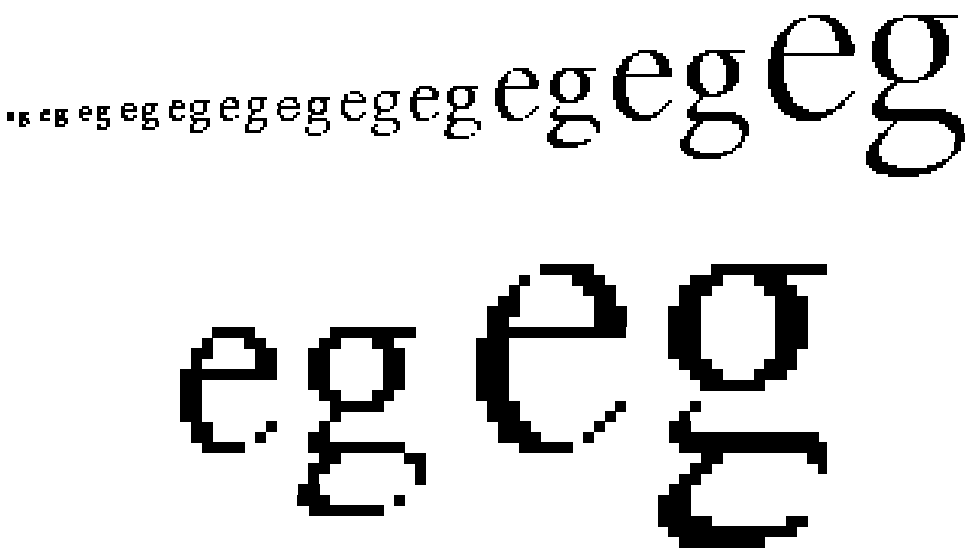

Figure 2. Top: a sequence of Times from 72 pp down to 6 pp at 72 dpi, showing dropouts at intermediate sizes; bottom: two enlarged pairs

another both horizontally and vertically. Upon regularising the outlines, this offset remains slanted down to moderately sized screenfonts, until it is eventually brought to a fully vertical or horizontal direction respectively. Although this attribute does assert a minimal thickness of stroke at the local extrema of curvilinear glyphs, in between these loci it can sometimes not avoid unfortunate patterns of jaggedness that result in a single missing pixel (example in Figure 2).

Clearly, missing pixels are intolerable, and equally clearly, a generic font representation prohibits any means of referring to pixels directly. The problem is intrinsic to the abstraction aimed at by a resolution-independent font representation: once the contour's knots are scaled, and possibly rounded, control is passed to the digitiser, with no further influence on the digitising process possible.

A mechanism is needed to serve as a stopgap for such glyphs. Unlike TrueType's *exception instructions* [7], which permit single pixels to be patched deliberately into the rastered glyphs, this mechanism should depend neither on a particular type size nor on a given resolution. Furthermore, it is important to restrict this measure to those glyphs which need it, and avoid burdening the other glyphs as well. This is an ideal case for making use of contour independence with *non-standard scan-conversion*.

The solution is to provide a contour class that consists of a pair of Bézier curves by itself, the inner and the outer outline of a curvilinear glyph. In this way, the contour class has the knowledge that the two outlines are related to one another. This contour class makes use of the standard scan-converter for Bézier curves, but instead of serializing the vertices of the scan-converted rectilinear region into the renderer, it records these digitised coordinate pairs. With that, the scan-converter can determine the locations at which pixels will be missing after rendering (filling). Since conceptually this is done in pixel space, it effectively realises a simple form of *image processing*.

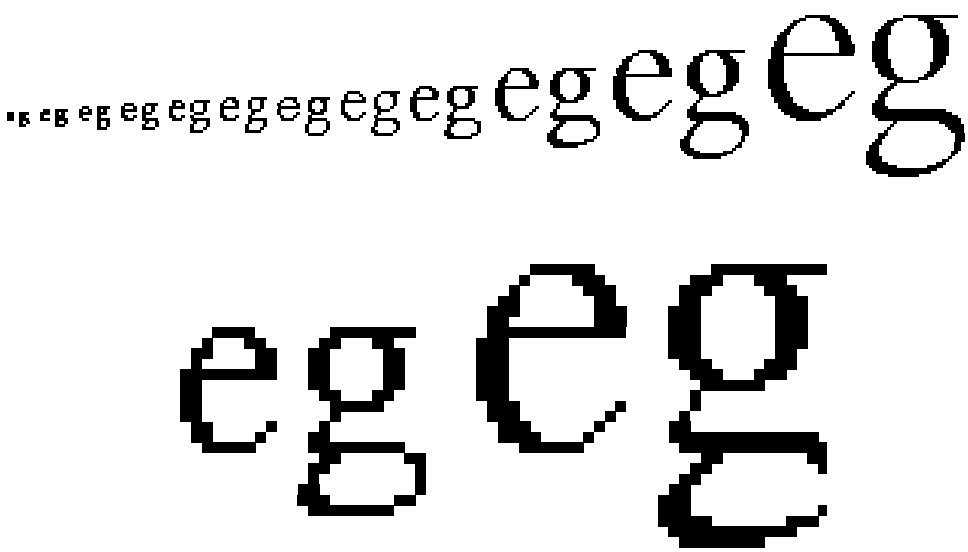

Figure 3. Top: the same sequence of Times eg *as in Figure 2, applying intelligent dropout prevention; bottom: the same two enlarged characters*

At this point the scan-converter knows about both the digitised and the original shape of the glyph. This allows the for intelligence to fix the resulting digitised image in accordance with the original shape of the contour to be packaged into the particular contour class (Figure 3). Subsequently, the digitised coordinate pairs are serialised into the renderer.

The resulting dropout control mechanism does not have to rely on a good guess as to which of two vertically or horizontally adjacent pixels to use as a stopgap. Rather, the respective *image-oriented algorithm* can fall back upon the original outline's topology. Together with the option of restricting the measure to absolutely necessary places, this particular use of OOP style constitutes a substantial advantage over other approaches.

3.2 Appearance-oriented filling

The boundaries of scan-converted continuous graphical objects often look more like irregular staircases than smooth contours. This jaggedness cannot be avoided, but sometimes its visible effects may be reduced without increasing the effective resolution. An example of this is *half-bitting*, a variant of anti-aliasing used on bi-level printers (Figure 4).

For the human eye half-bitting has the desired effect if the resolution is high enough such as not to let it easily tell apart individual pixels, although quantisation effects may still be visible. Commercial laser printers with a typical resolution of 300 to 600 dpi are good candiates for this technique. Toner dispersion further assists the desired effect. On screen the pixels are too coarse for this technique, there are too few pixels making up a character, and the pixel images do not spread.

The technique of half-bitting is not new at all, but the use of OOP style can encourage its integration into a font-scaler. The idea is to package the respective intelligence into a rendering class, rather than a contour class as in the previous section. Half-bitting may be

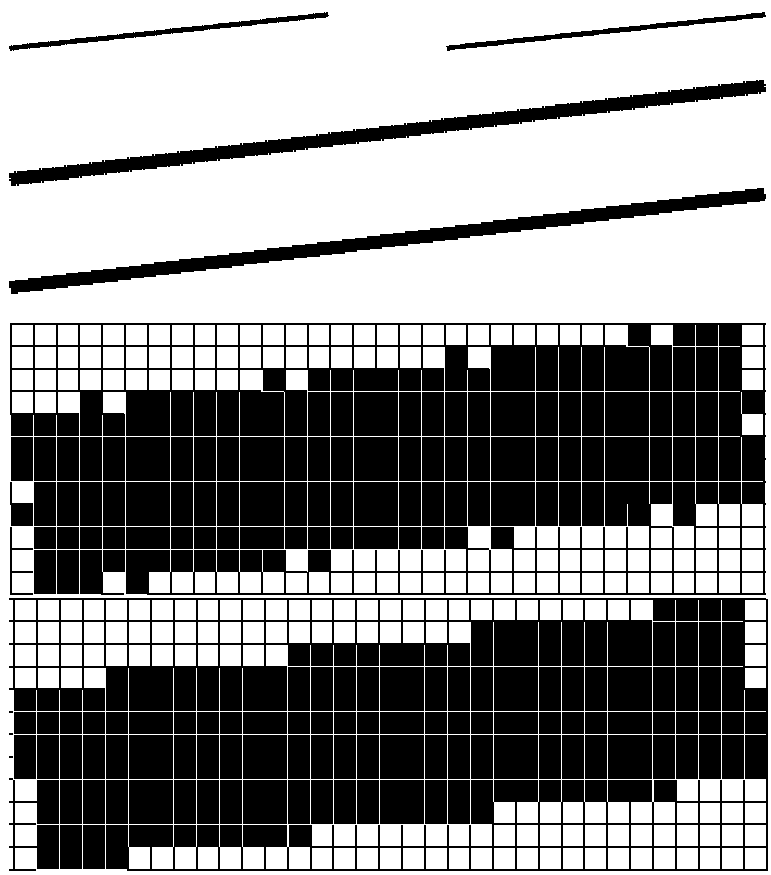

Figure 4. Top: a diagonal stroke at 300 dpi, filled with (left) and without (right) half-bitting. Middle: the same diagonals, enlarged (with, then without, half-bitting). Bottom: the same diagonals, generously enlarged (idem)

desirable not only for straight lines, but also for curves of moderate curvature. Analogous to the earlier example, the rendering class delays the actual rendering, recording the most recent part of the scan-converted rectilinear region. With this information lead, the renderer can detect those parts of the digital boundary which have a predominant direction for a certain length, and which are broken by only one unit step across.

Such elementary configurations are transformed into pieces of digital boundaries that reflect the demands of half-bitting. The transformed pieces have this step across advanced, followed by a step back, and finally another one across (Figure 5). Subsequently, the vertices of the transformed pieces of the scan-converted rectilinear region can be forwarded to the actual renderer in charge of the filling.

Significant advantages for the resulting half-bitting mechanism result from using OOP style. The mechanism is completely decoupled from both the digitising and the actual rendering. Therefore, neither the scan-converting algorithms nor the filling algorithm are subject to modification. Furthermore, the mechanism can be restricted to those devices on which this kind of *appearance-oriented filling* is most effective, while avoiding its

Figure 5. A single step across the digital boundary (left) is replaced by a meander (right)

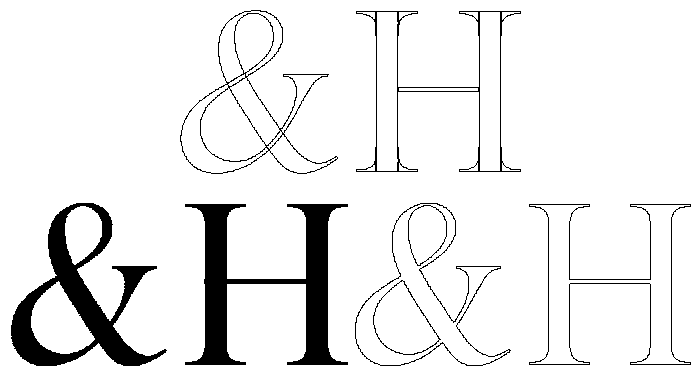

Figure 6. Times &H *(top), showing self-intersections and revealing the blueprints, rendered in filled (bottom left) and outlined (bottom right) style*

application to low-resolution screens. Much like the scaling factor, the *rendering tool* is becoming an interchangeable parameter of the font-scaler.

3.3 Smart outlining

To preserve the regularity properties of a font under scaling, characters are assembled out of pre-rastered glyphs. To render such glyphs without alteration as outlined or filled objects, intelligent rendering tools are decoupled from the pure scan-conversion. However, more demanding applications may wish to stroke the outline of entire characters, rather than their individual glyphs only. Unfortunately, outlining glyphs separately reveals the side-effects of the characters' blueprints (Figure 6).

Readers familiar with QuickDraw's *regions* [8] and PostScript's non-zero winding number *fill* command [9] will figure out easily how to combine these two ideas for an algorithm that eliminates adjacent, overlapping, or self-intersecting parts of glyphs, regardless of the number of components the characters are made of. Therefore, the actual algorithm is not subjected to further analysis here.

What is again important to emphasise is the possibility of packaging this kind of *smart outlining* into an intelligent rendering tool. Depending on the particular way these tools are programmed, and depending on the programming language's features regarding forwarding and delegation, it may be possible to combine half-bitting with smart outlining even without further programming! OOP style definitely helps the implementor of a font-scaler with the task of mastering the complexity of the given problem.

3.4 Unorthodox knots and numbers

Future work might make increased use of the inherent extensibility of knots and numbers. An extended knot class can refer to a pair of not necessarily collinear vectors that specify a

measuring rod and a direction of rounding. With that, the extended knot can assume a role similar to TrueType's *projection vector* and *freedom vector* [7].

Analogously, derived knots could satisfy simple *constraints* easily, such as informally 'knot k is half-way between this reference point and that vertex'. Constraining functions could be accomplished by derived number classes as well. These functions might go well beyond the safeguarding from vanishing under coarse scaling. Notice, however, that many font-specific constraints, such as are proposed in [10] and [11], are already an intrinsic part of the approach presented here.

Much as with the extensible contour classes, this provides both *knot* and *number independence*. What should be kept in mind, though, is the following warning: the ease with which derived component classes can be integrated into the framework should not lead to an abundance of mostly similar extensions. In particular, extensibility should not be misused to bypass the device-independence targeted by generic font representations. In our opinion, this would unveil a lack of comprehension of the matter at issue.

4 CONCLUSIONS

In this paper we advocate object-orientation to integrate separate solutions to different aspects of font-scaling into a simple and sound framework. The small base system is not limited, for extensions can be and have been smoothly integrated *ad hoc*. By mixing and matching orthogonal extensions, this does not entail a giant and unmanageable software package, despite the gamut of possible combinations.

Subsections 3.1 and 3.2 in particular have illustrated algorithms with a certain degree of *image-orientation*, as opposed to the predominant outline-orientation of the approach. This is intended to address the demand for talking not only about outlines, but also about appearance. Gutknecht and Kohen [12] conclude that image-oriented algorithms may yield aesthetically better results at low resolution, but that contour-oriented algorithms are generally faster. To a certain extent, we combine the strengths of both types of font-scaling approaches, without inheriting their weaknesses at the same time.

With one eye on Adobe's *hints*, which suggest the nature of a special-purpose contour or glyph [13], we might talk about *hint-independence*. With the other eye on TrueType's *instructions*, which operate on knots [7], we might talk about *instruction-independence* as well. With both contour *and* knot independence, we can package individual strategies into contours *or* knots, whichever is more appropriate for tackling a particular aspect of font-scaling. We consider this a major advantage over present font-scalers.

Further benefits of OOP style, such as in windowing systems and extensible editors, are usefully profited from. For instance, a variant of the editor proposed in [14] is used to provide versatile graphical feedback within the formally specified hierarchical decomposition of characters into their constituent parts. However, these benefits are not considered specific to font-related problems, and hence are not subjected to further analysis here. The primary lesson to be learnt from the use of OOP style in a font-scaler is the following: once we can no longer reduce the complexity of a given problem, then we should provide a means of mastering it by concept, rather than industriousness.

We are concerned to make clear that the examples presented in Section 3 are far from being new ideas, except maybe the non-standard scan-conversion. However, we think that the way in which object-orientation and extensibility are used, in particular regarding font technology integration, is new. Of course we assume that today's font-scalers follow the

line of structured programming and break their implementation into some form of building blocks, but with OOP style we can do this by concept, rather than by diligence. This saves a lot of programming time, program code, and maintenance costs.

Our OOP style approach yields a small font-compiler and font-scaler (or interpreter): less than 1000 Oberon-2 statements [15], and less than 16 KB of NS32*x*32 code! Typical extensions will contribute perhaps two more pages of source code each, or around 2 KB of object code. Yet OOP style does not reduce performance substantially. Our prototype (not optimised) compiles a font of average complexity (Times) into 0.25 to 1 KB of font representation per character. The interpreter generates 15 to 30 bitmapped characters per second (10 to 14 pt at 72 dpi, NS32532 at 25 MHz).

ACKNOWLEDGEMENTS

Since the present paper has been realised as a part of my font-design project [16], I am indebted particularly to the project supervisor Prof. Dr. J. Gutknecht. Quite a few ideas in this paper are influenced by the research done by colleagues at the institute, notably Dr. C. Pfister and Dr. C. A. Szyperski, and by our common activities in teaching OOP style. Many thanks finally go to my colleague M. Hausner for his competent proof-reading and constructive criticism of the contents of this paper. Last but not the least, my special thanks go to R. Southall and J. André for their patient help with the final edition.

REFERENCES

1. Beat Stamm, ‘A formalism for hierarchical outline-fonts’, in *Advanced Session of the IIEEJ*, Kogakuin University, Tokyo, Japan (1992).
2. Beat Stamm, ‘Dynamic regularisation of intelligent outline-fonts’, *Electronic Publishing: Origination, Dissemination, and Design*, **6**(3) (1993) (these proceedings).
3. Clemens A. Szyperski, *Insight ETHOS: On object-orientation in operating systems*, Ph.D. dissertation, Department of Computer Science, Swiss Federal Institute of Technology, Zürich, Switzerland, (1992).
4. Beat Stamm, ‘Algorithms for drawing thick lines and curves on raster devices’, Technical Report 107, Department of Computer Science, Swiss Federal Institute of Technology, Zürich, Switzerland, (1989).
5. Eliyezer Kohen, *Two-dimensional graphics on personal workstations*, Ph.D. dissertation, Swiss Federal Institute of Technology, Zürich, Switzerland, (1988). No. 8719.
6. Beat Stamm, ‘An object-oriented graphical tool box’, in *Third Eurographics Workshop on Object-Oriented Graphics*, Champéry, University of Geneva, Switzerland, (1992).
7. Apple Computer Inc., *The TrueType font format specification*, Cupertino, CA, 1990.
8. Apple Computer Inc., *Inside Macintosh*, Addison-Wesley, Reading, MA, 1985.
9. Adobe Systems Inc, *PostScript language reference manual*, Addison-Wesley, Reading, MA, 1985.
10. Jakob Gonczarowski, ‘Fast generation of unfilled and filled outline characters’, in *Raster Imaging and Digital Typography*, eds. Jacques André and Roger D. Hersch, pp. 97–110. Cambridge University Press, (1989).
11. Roger D. Hersch and Claude Bétrisey, ‘Advanced grid constraints: performances and limitations’, in *Raster Imaging and Digital Typography II*, eds. Robert A. Morris and Jacques André, pp. 190–204. Cambridge University Press, (1991).
12. Jürg Gutknecht and Eliyezer Kohen, ‘An analysis of font-scaling algorithms’. Presented at RIDT89, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1989.
13. Adobe Systems Inc., *Adobe Type 1 font format*, Addison-Wesley, Reading, MA, 1990.

14. Clemens A. Szyperski, 'Write-ing applications: Designing an extensible text editor as an application framework', in *Proc. TOOLS'92*, Dortmund, Germany, (1992).
15. Hanspeter Mössenböck and Niklaus Wirth, 'The programming language Oberon-2', *Structured Programming*, **12**(4), (1991).
16. Beat Stamm, *A hybrid approach to medium- and low-resolution font- scaling*, Ph.D. dissertation, Department of Computer Science, Swiss Federal Institute of Technology, Zürich, Switzerland (1994) (to appear).